Computer Science Department

TECHNICAL REPORT

SOME OBSERVATIONS CONCERNING FORMAL DIFFERENTIATION OF SET-THEORETIC EXPRESSIONS

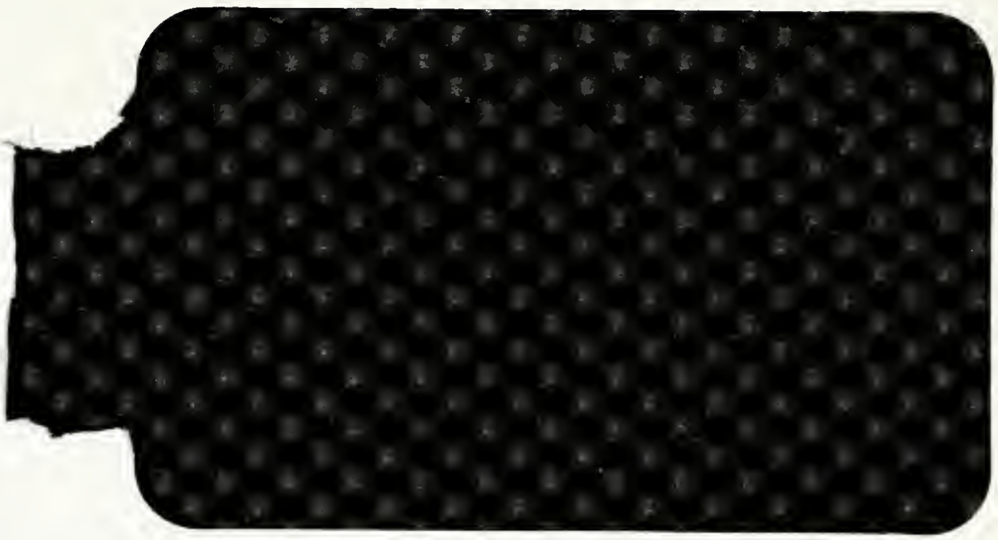
BY

MICHA SHARIR

OCTOBER 1979
REPORT NO. 016

NEW YORK UNIVERSITY

Department of Computer Science
Courant Institute of Mathematical Sciences
251 MERCER STREET, NEW YORK, N.Y. 10012



SOME OBSERVATIONS CONCERNING FORMAL DIFFERENTIATION OF SET-THEORETIC EXPRESSIONS

BY

MICHA SHARIR

OCTOBER 1979

REPORT NO. 016

* "This material is based upon work supported by the National Science Foundation under Grant No. NSF-MCS-78-18922.

Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the National Science Foundation."

ABSTRACT

This paper considers a variety of matters related to formal differentiation. We first suggest an algebraic approach to formal differentiation of a class of set-theoretic expressions. Then we go on to discuss the application of formal differentiation to loop fusion. Finally we apply formal differentiation to optimization of incremental construction of composite objects satisfying a given predicate. The techniques developed are illustrated by transformational construction of a variety of algorithms.

1. INTRODUCTION

The introduction of very high level languages such as SETL [De] which include set-theoretic constructs that can lead to computation of rather complicated set expressions has created new opportunities for program optimization, especially because elimination or improvement of certain basic constructs in this language, such as set union, set construction, iteration etc., is likely to have a significant pay-off in program execution time and storage requirements.

One such high-level optimization technique , which, following the terminology of Paige and Schwartz [PS], we will refer to as 'formal differentiation', was originally proposed by Earley [Ea], who called it 'iterator inversion'. This technique generalizes the classical 'reduction in strength' optimization technique used for languages of the FORTRAN level, to set-theoretic expressions. Its basic idea is to replace repeated costly computations of set theoretic expressions whose arguments change only slightly between successive computations by computations of 'incremental' or 'differential' expressions which are less expensive to evaluate, and which can be used to update the value of the original expression.

This technique has been studied by Fong and Ullman [FU], [Fo] and by Paige and Schwartz [PS], [Pa], who (at least currently) regard it as a technique somewhat too sophisticated to allow automatic treatment, but one which is systematic enough to admit a semi-automatic implementation. (A sketch of a possible implementation is described in [Pa].)

As developed by these authors formal differentiation turns out to be a powerful mechanism for improvement of a very high level version of an algorithm. Its application can produce a new version of an algorithm that runs an order of magnitude faster than its original version. These two properties of formal differentiation, namely that on one hand it is capable of changing the asymptotic behavior of algorithms, and on the other hand that it is more formalizable and systematic than many other program transformation methods, makes this a technique of central research interest. Many interesting examples of algorithm transformation by formal differentiation are given in [Pa].

The present paper contains three observations concerning formal differentiation. The starting point of our first observation, which is developed in section 2, is that as described till now (e.g. in [Pa]), formal differentiation techniques remain somewhat less methodical than their algebraic counterparts since they rely on larger collections of special rules. To remedy this, we will suggest a relatively simple formal procedure which, though not as general as Paige's rules, is more algebraic in flavor and can handle formal differentiation of a wide family of set theoretic expressions. A second observation, developed in section 3, suggests a new application of formal differentiation to loop fusion, which accomplishes fusion essentially by inserting one loop into another and then avoiding repeated elaborations of the inserted loop by applying formal differentiation to the expressions computed in it. A third observation, developed in section 4, shows how formal differentiation can be used to restrict and control the incremental construction of composite objects satisfying a given predicate.

Existing applications of formal differentiation aim at converting repeated computations of high-level set theoretic expressions to an incremental form; in section 4 we will study a situation where a composite object S is already being constructed incrementally, but in an inefficient manner, and show how this incremental construction can be substantially improved by analysis of certain set-valued expressions monitoring the incremental construction of S. All these observations are illustrated by examples, including a systematic derivation using loop fusion techniques of (a significant part of) an efficient garbage collection algorithm due to Dewar and McCann [DM].

The notations used in this paper closely follow those of the SETL programming language (cf. [De]), with certain deviations to enhance notational succinctness. For the sake of completeness we summarize these notations: Set union is written as A + B, intersection as A * B, set difference as A - B, symmetric difference as A sym B, set complementation as Ac, Cartesian product as A x B, set membership as X in A (the converse relation is denoted as 'notin'), set inclusion as A subset B, deterministic selection of an arbitrary element from a set as arb A, nondeterministic selection as arb* A, the null set as {}, the (multi-valued) map range construct as F[A] (so that F[A] is the range of F on the set A), inverse map application as F-1[A], the existential quantifier as 'exists X st P(X)' or as 'exists X in A st P(X)', the universal quantifier as 'forall X : P(X)' or as 'forall X in A : P(X)' and the standard logical connectives as 'and', 'or', 'not', 'implies' etc. We abbreviate A + {X} as A with X, and A - {X} as A less X. Tuples are denoted using square brackets, and compound operators are written as .op./ tuple of arguments, so that for example +/ [A(i) : i in [1...n]] denotes summation of A(1) through A(n). Assignment is denoted using ':=', but we abbreviate 'X := X .op. Y' as 'X .op.:= Y'. Note finally that linear notation, without subscripts or superscripts, is used in this paper.

I would like to express my gratitude to Jacob Schwartz for his help in preparing this manuscript, to Robert Dewar and Elia Weixelbaum for providing a derivation of the garbage collection algorithm discussed in section 3 and for stimulating discussions concerning that algorithm, to Robert Paige for a variety of illuminating comments, and to Keith Deak and Lambert Meertens for reviewing this paper.

2. A SIMPLIFIED TECHNIQUE FOR DIFFERENTIATING SET THEORETIC EXPRESSIONS

In this section we focus our attention on the problem of regularizing the formal differentiation mechanism described in [Pa] for the differentiation of certain set-theoretic expressions. To this end, we note an analogy between formal differentiation and algebraic differentiation, and begin with the following observation: Both in ordinary and in set-theoretic formal differentiation we start with a

function F(S) of an argument S, whose value at a particular S0 is known. Suppose that we change S 'slightly' to a new value S0 .op. DS; how can we express F(S0 .op. DS) as a corresponding 'slight' modification of F(S0)? In algebra, the infix operator .op. will be an addition, so that we can write

F(S0 + DS) = F(S0) + DF,

which implies

DF = F(S0 + DS) - F(S0)

and once this point is reached we can proceed to simplify the right-hand side by the standard rules of algebra to obtain a simpler form for DF.

The situation is similar for set-valued expressions. However, as treated by Paige and Schwartz, the small change DS and the corresponding difference DF are added or subtracted from the sets in question, and since in the set-theoretic case these operations do not have an inverse, we arrive at an equation like

F(S0 + DS) = F(S0) + DF,

whose solution is not obvious and must somehow be guessed, mainly by using distributive properties of F.

A simple remedy to this problem is available which makes use of the observation that there exists a binary operation on sets, namely - symmetric difference, which, taken together with set intersection, makes the class of all sets into a ring. If one takes symmetric difference (denoted below as 'sym') rather than union or set difference as the operation by which S and F are modified, then one is able to express DF explicitly, since the equation

(1) F(S0 sym DS) = F(S0) sym DF

can be transformed into

(2) DF = F(S0 sym DS) sym F(S0)

which then can be simplified using standard set-theoretic rules. This approach has the advantage of allowing any set-valued expression to be differentiated formally (with respect to any change DS in its set argument S). The question of whether it is profitable to do so is thereby detached from the actual differentiation, and becomes a matter of how much (2) can be simplified. (In the worst case, where (2) can not be simplified at all, computation of F(S0 sym DS) using (1) and (2) will be roughly three times more expensive than a direct computation.) This is very useful in the design of a semi-automatic transformation system, since it allows us to differentiate expressions formally without having to verify any specific enabling condition.

Simplification of (2) can of course be performed manually. However, as in calculus, it is possible to develop a set of rules for

computation of 'derivatives' of typical set expressions, and combinations of these rules can then be used to compute derivatives for a larger class of expressions. These differentiation rules include:

(3) D(A sym B) = DA sym DB;

(4) D(A * B) = A * DB sym B * DA sym DA * DB;

(5) D(A + B) = D(A sym B sym A * B) =
  DA sym DB sym A * DB sym B * DA sym DA * DB;

(6) DK = {}, if K is independent of the set being changed;

(7) D(Ac) = D(U sym A) = DU sym DA = {} sym DA = DA
  where Ac denotes the complement of A, and where U is the universal set;

(8) D(A x B) = A x DB sym DA x B sym DA x DB;

(9) D(F-1[A]) = F-1[DA]
  (inverse single-valued map application).

Note that both insertion of new elements into S and deletion of old elements from S are given a uniform treatment as special cases of symmetric difference.

This initial set of rules allows us to differentiate a variety of set expressions, in particular all those which do not involve quantification or any other reference to elements of their argument sets. While this initial family of differentiable expressions is still too limited to be usable in all significant cases, it is nevertheless interesting to see a few examples showing how such expressions can be differentiated.

To this end we assume that the set S is to be slightly modified, i.e. replaced by S sym DS. As a first example, let F(S) be the set-former expression

(10) F(S) = {X in S : P(X)},

where P(X) does not depend on S. This can be rewritten as

F(S) = S * P' ,

where P' denotes the truth-set of the predicate P. Since P' is independent of S, we obtain immediately

DF = DS * P' = {X in DS : P(X)} .

Thus, a program fragment of the form

(while {X in S : P(X)} /= E)

```
        S := S sym DS;
    end while;
```

can be transformed into

```
    FS := {X in S : P(X)};
    (while FS /= E)
        FS := FS sym {X in DS : P(X)};
        S := S sym DS;
    end while;
```

in a straightforward manner. Note that we do not care whether the change D(FS) to FS is incremental or decremental. However, in this particular case D(FS) can be shown to be of the same kind as DS, so that we can substitute set union (or subtraction, or both) for the 'sym' operator in both assignments in the loop. This observation is of interest because in some more complex cases it may only be possible to formally differentiate an expression (i.e. by a simplified derivative) if the change to S is incremental (or decremental).

As a slightly more complicated example, consider the following expression, which could appear, e.g., in a computation of the transitive closure of a single-valued map F:

```
(11)  G(S) = {X in S : F(X) notin S}
```

This can be rewritten as

```
    G(S) = S * F-1[Sc]
```

Applying our rules we obtain

```
    DG = DS * F-1[Sc] sym S * D(F-1[Sc]) sym DS * D(F-1[Sc])

       = DS * F-1[Sc] sym S * F-1[DS] sym DS * F-1[DS]
```

To give a somewhat more 'real' flavor to this formula, let us assume that S is augmented by a single element U. Then the last equation simplifies to

```
    DG = {U} * F-1[Sc - {U}] sym S * F-1{U}
```

It can now be seen that the first operand in this equation is to be added to G, whereas the second operand is to be subtracted from G. After a few more simplifications, we arrive at the following code:

```
    G := G - S * F-1{U};
    S := S with U;
    if F(U) notin S then
        G := G with U;
    end if;
```

As a third example, consider the program fragment

```
(while exists X in S, Y in S st P(X, Y))
    S with:= U;
end while;
```

In this case we can rewrite the while statement as

```
(while (S x S) * P' /= {})
```

which invites the differentiation of

```
(12)  H(S) = (S x S) * P'
```

Using the preceding rules, we obtain

```
DH = D(S x S) * P'

   = ( DS x S sym S x DS sym DS x DS ) * P'

   = ({U} x S) * P' sym (S x {U}) * P' sym ({U} x {U}) * P'
```

Again it is easy to see that all operands in the last equation are to be added to H. The update rule for H will then simplify to

```
H := H +
     {[U, Y] : Y in S st P(U, Y)} +
     {[X, U] : X in S st P(X, U)};
if P(U, U) then H := H with [U, U]; end if;
```

A closer look at the last example will reveal the fact that maintenance of the set H, even in this differential manner, may be superfluous, because all we really want to know is whether H is nonempty. This opens up the issue of formal differentiation of predicates involving slightly changing set arguments. Proper treatment of this issue can widen the class of formally differentiable expressions considerably, and with the addition of a few more rules which handle differentiation of expressions involving set cardinality, power sets, function spaces etc. we can come to rules which are gratifyingly general.

To this end, let P(S) denote a predicate involving some set S, and assume that S is to be changed into S sym DS. We would like compute P(S sym DS) using the value of P(S) which we assume that we already have calculated. To do so, we can write

```
(13)  P(S sym DS) = P(S) sym DP ,
```

where we extend the symmetric difference operator to act on boolean values in an obvious manner (i.e. P sym Q is true iff exactly one of P and Q is true). Then we can write

```
(14)  DP = P(S sym DS) sym P(S)
```

and again face the problem of simplifying this expression. For simplicity let us assume that the only operators appearing in P are standard set theoretic operators such as union, intersection, complementation, inclusion etc. Since quantifiers can be changed into equalities or inequalities involving set arguments, we assume that no quantification appears in P.

It follows that P(S) can be represented as disjunction and conjunction of primitive units, each having the form 'A = {}' or its negation, where A is some set-valued expression.

To derive a set of rules for formal differentiation of predicates we first note that rules (3) - (7) extend easily to the boolean case, to wit

(15) D(P sym Q) = DP sym DQ;

(16) D(P and Q) = (P and DQ) sym (Q and DP) sym (DP and DQ);

(17) D(P or Q) =
DP sym DQ sym (P and DQ) sym (Q and DP) sym (DP and DQ);

(18) DR = false,
where R is independent of the set being changed;

(19) D(not P) = DP.

These rules reduce the problem of differentiating a predicate of the form just considered to that of differentiating a predicate having the basic form

(20) P(S) = (F(S) = {}) ,

where F(S) is some set-valued expression depending on S.

Using (14), we can write

(21) DP = (F(S sym DS) = {}) sym (F(S) = {})

= ((F sym DF) = {}) sym (F = {})

= (F sym DF = {} and F /= {}) or
(F = {} and F sym DF /= {})

= DF /= {} and (F = {} or F sym DF = {})

= DF /= {} and (P or F = DF)

As it stands, the final form of (21) does not allow us to maintain P(S) in a purely differential manner because we may have to maintain F(S) as well in order to compute subpredicate (F = DF). But there is a case in which a better form is available. Suppose that we can show that F = DF only if F = {}. This will be the case e.g. if DF is known to be disjoint from F. Then (21) simplifies to

$$(22) \quad DP = (P \text{ and } (DF /= \{\}))$$

so that by (13)

$$(23) \quad P(S \text{ sym } DS) = P(S) \text{ sym } (P(S) \text{ and } (DF /= \{\}))$$

$$= P(S) \text{ and } (DF = \{\})$$

as might be expected. This formula has the pleasant property that when it applies F need not be maintained at all; only its derivative need be computed.

Similar formulae can be developed for predicates of the form

$$(24) \quad Q(S) = (F(S) /= \{\})$$

Indeed, by (19), DQ = D(not Q), so that we can use (21) to compute DQ. Again, if we can show that F = DF only if F = {}, we can obtain

$$(25) \quad Q(S \text{ sym } DS) = Q(S) \text{ or } (DF /= \{\})$$

The formulae (23) and (25) are particularly useful when the predicates in question control a loop within which formal differentiation is desired. For example, consider the case

```
(while F(S) = {})
    S with:= U;
end while;
```

Here the predicate P(S) = (F(S) = {}) is always true at the point where we want to differentiate it. Thus if DF = F at that point it follows that DF = {}, so that we can use (23) to deduce that P(S sym DS) = (DF = {}).

Let us now return to consideration of (21), but now assume that we are unable to eliminate the nondifferential subpredicate F = DF. This being so, we will have to maintain the set F during loop execution (using formal differentiation if possible) and consequently compute P each time from its definition (22) without using (21) at all.

As an important example in which F will have to be maintained, consider the loop

```
(while F(S) /= {})
    S with:= U;
end while;
```

Since we want to differentiate F inside the loop, we know that at the update point F /= {}. We claim that in this case the equality DF = F need not (and in general will not) imply F = {}, so that we can not maintain the predicate Q(S) = (F(S) /= {}) in a differential manner. Indeed, if we had DF = F only if F = {}, this would imply by (25),

Q(S sym DS) = Q(S) or (DF /= {}) = Q(S) = true,
which implies that the loop will never terminate since the test in the 'while' statement will always be true. Hence in this case we must allow for the possibility that DF = F when F /= {}, which does not allow us to maintain Q differentially. (In fact this possibility is equivalent to loop termination, provided that the loop is not bypassed.) Consequently, we must maintain the set F (hopefully in a differential manner) and leave the 'while' condition unchanged.

For an overall demonstration of the power of our formalism , we now consider the following transitive closure schema (in which the actual selection of the next element U is not shown):

```
      S := S0;
      (while exists X in S st not (F{X} subset S) )
          S with:= U;      $ where U notin S
      end while;
```

Here we want to differentiate the predicate

```
      P(S) = exists X in S st not (F{X} subset S)
```

which can be rewritten as

```
      P(S) = exists X in S, Y in Sc st Y in F{X}

           = exists [X, Y] in S x Sc st Y in F{X}

           = (S x Sc) * A' /= {}
```

where

```
      A' = {[X, Y] : Y in F{X} }    ( = F)
```

Let H(S) = (S x Sc) * A'. Then we have

```
      DH = (S x DS sym DS x Sc sym DS x DS) * A'

         = (S x {U} sym {U} x Sc sym {U} x {U}) * A'
```

As noted above, in such situations H must be maintained. Thus we arrive at the following formally differentiated version:

```
      S := S0;
      H := {[X, Y] : X in S, Y in Sc st Y in F{X}};
      (while H /= {})
         H := H
            - {[X, U] : X in S st U in F{X}}
   $ subset of H
            + {[U, Y] : Y in (Sc less U) st Y in F{U}};
   $ disjoint from H
         S with:= U;
      end while;
```

REMARK: To obtain the well known 'workset' algorithm for transitive

closure from the above version, all we have to do is to maintain the set RANGH = range H instead of H itself. This is possible since the predicate H /= {} is equivalent to the predicate RANGH /= {}. The main difficulty in accomplishing this transformation is to show that subtraction of the first set from H has the same effect as subtracting {U} from its range. Once this is proved, we can eliminate H and so arrive at the following version

```
S := S0;
RANGH := {Y : X in S, Y in Sc st Y in F{X}};
(while RANGH /= {})
    RANGH := RANGH less U
        + {Y : Y in F{U} st Y notin (S with U)};
    S with:= U;
end while;
```

which is precisely the standard workset algorithm, provided that one always selects U from the current RANGH set. The decision to select U from RANGH is quite natural, as it will cause the removal of U from that set, and so 'help' it to diminish to the null set, which is our goal. This observation has general significance, and will be developed more fully in section 4 below.

For a second, more complicated example, consider the following problem: Given a set E and a subset A of E x E, find the smallest equivalence relation S on E which contains A. This problem can be reformulated as follows (where the minimality condition is temporarily ignored):

```
find S a subset of E x E such that
    A subset S and
    (forall X in E : [X, X] in S) and
    (forall X in E, Y in E : [X, Y] in S implies [Y, X] in S) and
    (forall X in E, Y in E, Z in E :
            [X, Y] in S and [Y, Z] in S implies [X, Z] in S)
```

Using a standard scheme for the construction of a subset of a given set subject to a given constraint by adding to it one element at a time, we obtain the following 'implementation' of this specification:

```
S := {};
(while not (A subset S) or
        (exists X in E st [X, X] notin S) or
        (exists X in E, Y in E st
            [X, Y] in S and [Y, X] notin S) or
        (exists X in E, Y in E, Z in E st
            [X, Y] in S and [Y, Z] in S and [X, Z] notin S))

    select U := [V, W] nondeterministically st U notin S;
    S with:= U;
end while;
```

Next we differentiate the expression appearing in the 'while' statement. To do so, we rewrite this expression as

```
A * Sc /= {}  or
{[X, X] : X in E} * Sc /= {}  or
{[X, Y] : X in E, Y in E st
    [X, Y] in S and [Y, X] notin S} /= {}  or
{[X, Y, Z] : X, Y, Z in E st
    [X, Y] in S and [Y, Z] in S and [X, Z] notin S} /= {}
```

This can be further simplified as follows: Let B denote the set {[X, X] : X in E}. Introduce a few mappings as follows:

```
I([X, Y]) = [Y, X];
F([X, Y, Z]) = [X, Y];
G([X, Y, Z]) = [Y, Z];
H([X, Y, Z]) = [X, Z];
```

Then the above predicate can be rewritten as

```
A * Sc /= {}  or
B * Sc /= {}  or
S * I-1[Sc] /= {}  or
F-1[S] * G-1[S] * H-1[Sc] /= {}
```

and better still as

```
(A * Sc + B * Sc + S * I-1[Sc] +
    F-1[S] * G-1[S] * H-1[Sc]) /= {}
```

which can be formally differentiated using the rules given above. The simplest procedure is to differentiate each set expression separately. Let us denote the four set expressions appearing in the above predicate as K(S), L(S), M(S), N(S) respectively. Then we have

```
DK = A * DS

DL = B * DS

DM = S * I-1[DS] sym DS * I-1[Sc] sym DS * I-1[DS]

DN = F-1[DS] * G-1[S] * H-1[Sc] sym
     F-1[S] * G-1[DS] * H-1[Sc] sym
     F-1[S] * G-1[S] * H-1[DS] sym
     F-1[DS] * G-1[DS] * H-1[Sc] sym
     F-1[DS] * G-1[S] * H-1[DS] sym
     F-1[S] * G-1[DS] * H-1[DS] sym
     F-1[DS] * G-1[DS] * H-1[DS]
```

Since DS = {U} = {[V, W]} subset Sc, we can simplify the above to

```
DK = if U in A then {U} else {} end

DL = if U in B then {U} else {} end

DM = if [W, V] in S then {[W, V]} else {} end  sym
     if [W, V] notin S then {[V, W]} else {} end  sym
```

```
        if W = V then {[V, W]} else {} end

    DN = {[V, W, Z] : Z in E st [W, Z] in S and [V, Z] notin S} sym
         {[X, V, W] : X in E st [X, V] in S and [X, W] notin S} sym
         {[V, Y, W] : Y in E st [V, Y] in S and [Y, W] in S} sym
         {[V, V, V] : V = W and [V, V] notin S} sym
         {[V, W, W] : [W, W] in S} sym
         {[V, V, W] : [V, V] in S} sym
         {[V, V, V] : V = W}
```

We have thus converted our algorithm into one which uses four different worksets each of which requires relatively little updating when S is augmented. The new version is as follows:

```
    S := {};
    K := A;
    L := B;
    M := {};
    N := {};
    (while K /= {} or L /= {} or M /= {} or N /= {})
        select U := [V, W] as before;
        K := K sym DK;
        L := L sym DL;
        M := M sym DM;
        N := N sym DN;
        S with:= U;
    end while;
```

This new version still has the generality of the original specification, in the sense that all sets satisfying the conditions of the specification (except for minimality), and only such sets will be constructed by (successful) executions of the above nondeterministic program. For methods which improve the way in which U is selected in the preceding version, see section 4.

5. LOOP FUSION USING FORMAL DIFFERENTIATION

In this section we describe an interesting application of formal differentiation to program construction. This technique, which has been called loop fusion, is applicable when the program contains two or more consecutive loops where the first loop builds up a certain composite object which is used in the second loop to build up another composite object. When this is the case, we may be able to insert the second loop into the first one so as to make the succeeding copy of the second loop redundant, and then, applying formal differentiation, to replace repeated executions of the inserted loop by incremental computations,

resulting in an effective fusion of the two loops. This approach has the advantage of being more general, more formal and more systematic than standard loop fusion techniques. It is especially applicable to high-level program variants in which the loops being fused are implicit in constructs such as set-formers, quantifiers, compound operators etc. Another connection between loop fusion and formal differentiation is noted in [Pa].

As a typical example of loop fusion by formal differentiation, consider the following code:

```
S := {};
(while not P(S))
     X := exp;
     S with:= X;
end while;

T := K(S);
```

Here the first loop constructs a set S satisfying P incrementally, while the following statement computes some set-theoretic expression K(S) which may require some iterations over S and related objects. This two-pass computation can be fused into one loop by moving the assignment T := K(S) into the first loop and its preheader, thereby making the assignment redundant on exit from the loop. This is possible since T is neither used nor modified within the first loop. Then, applying formal differentiation to K(S), we can replace repeated computations of K(S) within the first loop by incremental updating of T, which will give us the fusion effect we are after.

To give a more concrete example, consider the following program, in which we want to build up a map and then compute its inverse map:

```
S := {};
(while not P(S))
     [X, Y] := exp;
     S with:= [X, Y];
end while;

T := {[A, B] : [B, A] in S};
```

To fuse the two loops together, we insert computations of T into the end of the while loop and into its preheader, and, after eliminating the original computation of T which has now become redundant, obtain the following version:

```
S := {};
T := {};
(while not P(S))
     [X, Y] := exp;
     S with:= [X, Y];
     T := {[A, B] : [B, A] in S};
end while;
```

Formal differentiation of the expression for T will then yield

```
      S := {};
      T := {};
      (while not P(S))
          [X, Y] := exp;
          S with:= [X, Y];
          T with:= [Y, X];
      end while;
```

in which the required loop fusion is accomplished.

This technique can be generalized to handle other loop fusion patterns. An interesting case is that in which more than one computation depends on a set (or tuple) S. Such a program fragment might be something like the following:

```
      T1 := K1(S);
      T2 := K2(S);
      . . . .
      Tn := Kn(S);
```

Suppose now that S is a constant, or a set read by an input statement. In such cases there will be no explicit 'first' loop into which the computations of T1, T2 ... Tn can be inserted. Even in this apparently unfavorable case, we can create a loop which builds up S incrementally, and then use that loop as our fusion target exactly as before. As an ~~example~~, consider the following program:

```
      T1 := +/ [X**2 : X in [1 ... N]];
      T2 := +/ [X**3 : X in [1 ... N]];
```

This code can be transformed as follows:

```
      S := [];
      T1 := T2 := 0;
      (forall A in [1 ... N])
          S with:= A;
          T1 := +/ [X**2 : X in S];
          T2 := +/ [X**3 : X in S];
      end forall;
```

(Note that renaming of bound variables may be required during loop fusion.) Application of formal differentiation to the expressions for T1 and T2 will then yield

```
      S := [];
      T1 := T2 := 0;
      (forall A in [1 ... N])
          S with:= A;
          T1 +:= A**2;
          T2 +:= A**3;
      end forall;
```

Finally, noting that S is now dead in this code, we can eliminate it,

thereby accomplishing the required loop fusion.

Another more complex but still typical case is that in which several set-theoretic expressions, each depending on previously computed expressions follow each other. Such cases can be handled by exactly the same technique as above, that is, by fusing all these computations into a first loop (which, as in the preceding example, may have to be created explicitly) and then by applying formal differentiation to the expressions in question.

An interesting example of loop fusion using formal differentiation appears in the construction of a garbage-collection algorithm due to Dewar and McCann [DM]. Part of the transformational derivation of this algorithm is reconstructed below to demonstrate our loop fusion technique. For a reference to this derivation, see [DSW]. A very high-level specification of that algorithm reads as follows:

assume P is a given storage map, which maps each storage cell to a tuple of cell pointers. Each cell X in domain P designates some storage block and P(X) is a list of all cells to which X points. Only cells which can be reached from the first cell 0 are still active, and these should be compacted together, with pointer values properly adjusted.

The following slightly more detailed code expands the preceding description:

```
$ step 1: find all cells reachable from 0 (i.e. active cells);

      find U : subset domain P st 0 in U and
           (forall A in U, C in P(A) : C in U) and
           smallest(U, inclusion);

$ step 2: compute the new locations of active cells

      N := {[B, 0 +/ [#P(C) : C in U st C < B] ] : B in U};

$ step 3: compute the compacted storage map

      Q := {[N(C), [N(D) : D in P(C)] ] : C in U};

$ finally, re-assign Q to the original map P

      P := Q;
```

To optimize this, we first split step 3 into two substeps, one which only computes the range of Q, and another which computes the map Q itself. This leads to the following refinement:

```
$ step 3.1:

      K := {[C, [N(D) : D in P(C)] ] : C in U};

$ step 3.2:

      Q := {[N(C), K(C)] : C in U};
```

Next we fuse the computation of K with the computation of N, which is first expanded into a loop, and so obtain the following fragment:

```
$ steps 2 and 3.1:

        N := {};
        K := {[C, [OM : D in P(C)] ] : C in U};

        (forall B in U)
            T := 0 +/ [#P(C) : C in U st C < B];
            N(B) := T;
            K := {[C, [N(D) : D in P(C)] ] : C in U};
        end forall;
```

Next we differentiate the expression for K within the forall loop, relative to the augmentation of N. To do so, we note that when N is augmented by the assignment 'N(B) := T' K only changes at points C such that B is in P(C), and the change at these points amounts to changing all components of K(C), which correspond to components of P(C) equal to B, to T. This calls for maintaining a 'memo map' R which should map each B in domain N to all pairs [C, I] st P(C)(I) = B and C in U. This leads to the following fragment:

```
$ steps 2 and 3.1:

        N := {};
        K := {[C, [ ] ] : C in U};

        R := {[B, [C, I] ] :
                        C in U, I in [1 ... #P(C)] st P(C)(I) = B};

        (forall B in U)
            T := 0 +/ [#P(C) : C in U st C < B];
            N(B) := T;
            (forall [C, I] in R{B})
                K(C)(I) := T;
            end forall;
        end forall;
```

REMARK: Note the generality of the loop fusion technique exemplified by the above transformation. Standard techniques might try to fuse these loops by using the fact that both iterate over U, but this will fail as the computations required in the second loop for a particular C in U are not related at all to the computations performed for the same C in the first loop.

Next we apply loop fusion again, this time fusing the computation of R into the loop computing U. To this end we first expand the computation of U into a loop, using a standard workset-oriented transitive closure scheme. This gives the following fragment:

```
$ step 1 as a loop:
```

```
      U := {};
      W := {0};
      (while W /= {})
           A from W;
           U with:= A;
           W +:= {C in P(A) st C notin U};
      end while;
```

```
$ computation of R:

      R := {[B, [C, I] ] :
                  C in U, I in [1 ... #P(C)] st P(C)(I) = B};
```

Then, fusing the loops together, we obtain

```
$ step 1 and computation of R:

      U := {};
      W := {0};
      R := {};
      (while W /= {})
           A from W;
           U with:= A;
           W +:= {C in P(A) st C notin U};
           R := {[B, [C, I] ] :
                    C in U, I in [1 ... #P(C)] st P(C)(I) = B};
      end while;
```

Next the expression for R is differentiated. This is easily done and yields the following fragment:

```
$ step 1 and computation of R:

      U := {};
      W := {0};
      R := {};
      (while W /= {})
           A from W;
           U with:= A;
           W +:= {C in P(A) st C notin U};
           R +:= {[B, [A, I] ] :
                    I in [1 ... #P(A)] st P(A)(I) = B};
      end while;
```

At this point, a new loop fusion step becomes applicable, namely we can fuse the loop updating R with the loop updating W. To see this, we note that both loops depend on the tuple [1...#P(A)]. Consequently we can create an auxiliary loop whose sole purpose is to build up that tuple; then both computations can be inserted into that loop. After the required clean-ups, we obtain the following fragment:

```
$ step 1 and computation of R:

      U := {};
```

```
W := {0};
R := {};
(while W /= {})
     A from W;
     U with:= A;
     (forall I in [1 ... #P(A)])
          C := P(A)(I);
          R with:= [C, [A, I]];
          if C notin U then
               W with:= C;
          end if;
     end forall;
end while;
```

CONCLUSION: three loop fusion steps suffice to produce this version of the algorithm (still not its final version, which is obtained by applying additional improving transformations; cf. [DSW] for details), which might otherwise have been considered ingenious rather than systematically derivable.

4. FORMAL DIFFERENTIATION AND THE INCREMENTAL GROWTH OF SETS SATISFYING A GIVEN PREDICATE

In this section we will describe a way in which formal differentiation can be applied to control the manner in which set-theoretic objects are constructed from given coarse specifications. As observed in e.g. [Sh], a common and useful technique for the construction of composite objects satisfying a given specification is to build them incrementally. A typical example of such a process is the construction of a subset S of some universal set E which satisfies a certain predicate P(S). Using a notation suggested in [Sh], we write this specification as

```
find S : subset E st P(S);
```

This can be realized by using the following incremental construction scheme (where, as noted earlier, arb* denotes a nondeterministic selection of an element from a set):

```
S := {};
(while not P(S))
     X := arb* (E - S);
     S with:= X;
end while;
```

This scheme has the property that its successful executions yield exactly all subsets S of E satisfying P(S) which are sequentially

minimal with respect to the property P(S) (i.e. which are such that S can be enumerated as X1 ... Xn such that for all j < n P({X1...Xj}) is false). Thus transformation of the specification given above into this scheme is correctness preserving in the sense that the specification has a solution if and only if the scheme has a solution. Moreover, all minimal subsets S (or the smallest such S) satisfying P(S) will also be produced by the above scheme (if they exist).

However, in order to refine this scheme into a more efficient algorithm, it is of crucial importance to be able to select the next element X to be added to S in a 'better' way, either by making its selection deterministic, or else by limiting the search space for the (backtracked) selections of X in a significant manner. Heuristically we would like to select the next X not just to be any element in E - S, but rather want to select it in some profitable manner, e.g. because we can tell by the partial contents of S that X must eventually belong to a superset of S satisfying P, or because inserting this X into S will serve to decrease some termination function and hence guarantee (faster) convergence of the algorithm.

In this section we attempt to systematize this process of refinement of the search for S. To this end we state certain metarules which are applicable for a rather large class of such problems and which show how to refine the growth of S in a way which is based on the form of P(S) and which preserves algorithm correctness.

Let us assume that in the code shown above P(S) has the form 'K(S) = {}', where K(S) is some set-valued expression depending on S. As observed in [Sh], it is generally useful to formally differentiate K(S) with respect to the augmentation of S in the 'while' loop, rather than to compute it afresh in each iteration. If this is done we will come to the following version:

```
(*)     S := {};
        K' := K({});
        (while K' /= {})
             X := arb* (E - S);
             DK := DK(S, {X});
             K' := K' sym DK;
             S with:= X;
        end while;
```

As a concrete example, consider the transitive closure example examined in section 2. Its next-to-last version reads (after some renaming of variables) as follows:

```
        S := S0;
        K := {[U, V] : U in S, V in Sc st V in F{U} };
        (while K /= {})
             X := arb* (E - S);
             K := K
                  - {[U, X] : U in S st X in F{U}}
                  + {[X, V] : V in (Sc less X) st V in F{X}};
             S with:= X;
```

```
     end while;
```

(Note that, because S is initialized to a nonempty set, this is not exactly an instance of (*); see, however, an extended comment at the end of this section concerning this deviation.) In this example a better way to choose X is to insist that the choice of X cause elements to be removed from K, i.e. select X for which there exists U in S st X in F{U}. Moreover, in the example before us it is always possible to select such an X as long as K /= {} (in fact any element in range K will do). This refined selection will still yield the smallest set S satisfying the predicate 'K = {}'.

As already noted in section 2, the rationale for such improved selection is obvious heuristically: Since our goal is to reduce K(S) to the null set, we may as well attempt to remove elements from it each time we select a new element X to be added to S.

The applicability of this heuristic can be observed in all case studies considered so far in this paper and in [Sh]. This motivates the following definition which captures and generalizes the phenomenon noted above:

SERIAL SELECTABILITY: We say that the equation 'K(S) = {}' has a serially selectable solution (or equivalently that K(S) has the serial selectability property) if for each S subset E such that K(S) = {} and such that S is minimal with respect to this condition, there exists an enumeration X1...Xn of S such that for all j in [1...n]

```
     K({X1, ..., Xj-1}) not subset K({X1, ..., Xj})
```

or, in other words,

```
     DK({X1, ..., Xj-1}, {Xj}) * K({X1, ..., Xj-1}) /= {}
```

The significance of serial selectability is manifested in the following observation:

OBSERVATION: Serial selectability of K(S) implies that if the (nondeterministic) selection of X in the above scheme (*) is restricted so as to cause elements to be removed from the current set K' (but not further restricted) then one obtains a scheme which is equivalent to scheme (*) in the sense that it also constructs all minimal subsets S of E satisfying K(S) = {}. In particular, if there exists a smallest subset S of E satisfying K(S) = {} then this S will also be computable by the following modified scheme:

```
(**)  S := {};
      K' := K({});
      (while K' /= {})
           X := arb* {W in E - S st DK(S, {W}) * K' /= {}};
           DK := DK(S, {X});
```

```
        K' := K' sym OK;
        S with:= X;
    end while;
```

We will show below that the family of set expressions K(S) for which 'K(S) = {}' has a serially selectable solution is fairly large, and hope that further study can extend this family still further.

Serial selectability is a consequence of the following property:

INCREMENTAL SELECTABILITY: A set-valued expression K(S) is said to have the incremental selectability property if for each pair S, T of disjoint subsets of E such that T /= {} and

K(S) not subset K(S + T)

there exists X in T such that

K(S) not subset K(S + {X})

This is shown by the following

PROPOSITION 1: Incremental selectability implies serial selectability.

PROOF: Suppose that K has the incremental selectability property, and let S be a minimal subset of E satisfying K(S) = {}. If S = {} then the equation K = {} has a vacuously serially selectable solution. Otherwise we have K({}) /= {} and therefore

K({}) not subset K({} + S) = {}

so that by incremental selectability there exists X1 in S such that

K({}) not subset K({X1})

We can continue in this manner till all elements of S have been selected. Indeed, suppose that X1,X2...Xj have already been selected for some j<n. By the minimality of S we know that

K({X1...Xj}) not subset K({X1...Xj} + (S - {X1...Xj}))
    = K(S) = {}

and by the incremental selectability property there exists Xj+1 in S - {X1...Xj} such that

K({X1...Xj}) not subset K({X1...Xj+1})

which concludes the proof.

Q. E. D.

Next we state sufficient conditions for the incremental

selectability property to hold. Suppose that K(S) has the form
(1) K(S) = K1(S) + K2(S) + ... + Kr(S)

where K1 ... Kr are set-valued mappings having disjoint ranges, and where each Ki can be represented in the form

(2) Ki(S) = Ai(S) * Bi(S)

where Ai(S) is an arbitrary monotone increasing set-valued expression in S, and where Bi(S) is an arbitrary monotone decreasing set-valued expression in S.

(As a typical example, return to the transitive closure scheme described above. There we have

K(S) = {[U, V] : U in S, V in Sc st [U, V] in F}

= (S x Sc) * F = P1-1[S] * P2-1[Sc] * F

where P1, P2 are the projections of E x E onto its first and second components respectively. This expression K(S) has the form (1), and involves a single term having the form (2) with

A(S) = F * P1-1[S]

B(S) = P2-1[Sc])

THEOREM 2: Expressions K(S) of the form (1) have the serial selectability property provided that for each i <= r DBi(S, DS) is sub-distributive in the argument DS for DS disjoint from S (i.e. if Bi(S) is differentiated with respect to an increase of S). Here, a set-valued function F(A) of a set-valued variable A is said to be sub-distributive if

F(A1 + A2) subset F(A1) + F(A2)

PROOF: It is easily seen that it suffices to prove incremental selectability for each subexpression (2) of K. Applying our formal differentiation rules to such a subexpression Ki we obtain

DKi = Ai * DBi sym DAi * Bi sym DAi * DBi

However, assuming that the change in S relative to which this derivative is computed consists of addition of new elements (as is the case in scheme (**)) and using the monotonicity of Ai and Bi, we get

Ai * DAi = {}

DBi subset Bi

Hence, using the assumption concerning DBi which was made above, the expression

Mi(S, DS) = Ki(S) * DKi(S, DS) = Ai(S) * DBi(S, DS)

is sub-distributive with respect to the argument DS. That is,

Mi(S, T1 + T2) subset Mi(S, T1) + Mi(S, T2)

Incremental selectability of Ki(S) (and hence of K(S) itself) is now easy to establish. Indeed, suppose that S, T are disjoint subsets of E such that T /= {} and

Ki(S) not subset Ki(S + T)

Put DS = T; it follows that

Mi(S, T) = Ki(S) * DKi(S, DS) /= {}

But, using sub-distributivity,

Mi(S, T) subset +/[Mi(S, {X}) : X in T]

Hence there exists X in T such that Mi(S,{X}) /= {}. This implies incremental selectability for Ki(S) and our theorem then follows from proposition 1.

Q. E. D.

Let MSD denote the class of all monotone expressions B(S) having the property that DB(S, DS) is sub-distributive in DS if DS is disjoint from S. Let MSD+ denote those expressions in MSD which are monotone increasing, and let MSD- denote those which are monotone decreasing. The following lemma indicates that MSD- is a fairly large family of expressions:

LEMMA 3: MSD, MSD+ and MSD- have the following properties:

(a) The identity function B(S) = S is in MSD+;

(b) If B(S) is in MSD+ then B(S)c is in MSD-;
if B(S) is in MSD- then B(S)c is in MSD+;

(c) If B(S) is in MSD+ then F[B(S)] is in MSD+ for any map F;

(d) If B(S) is in MSD- then F-1[B(S)] is in MSD-, for any single valued map F;

(e) If B1(S), B2(S) are in MSD-, then B1(S) * B2(S) is in MSD-.

(f) Let P(X, S) be a predicate which is monotone increasing in S, i.e. whenever S subset T then P(X, S) implies P(X, T). Suppose moreover that

DP(X, S, DS) = P(X, S + DS) and not P(X, S)

is sub-distributive in DS for DS disjoint from S, i.e.

DP(X, S, DS1 + DS2) implies DP(X, S, DS1) or DP(X, S, DS2)

Then the function

    B(S) = {X : P(X, S)}

is in MSD+. An analogous result can be stated for monotone decreasing predicates.

(g) In particular, for any predicate Q(X, Y) (which is independent of S) we have

    E1(S) = {X : exists Y in S st Q(X, Y)}   is in MSD+

and

    B2(S) = {X : forall Y in S : Q(X, Y)}   is in MSD-

PROOF: Most of these assertions are trivial. To prove (c), we note that since B is monotone increasing DB is disjoint from B. Hence,

    DF[B] = F[B + DB] - F[B] = F[DB] - F[B]

from which sub-distributivity follows immediately. To prove (e), we have

    D(B1 * B2) = DB1 * B2 sym B1 * DB2 sym DB1 * DB2

               = DB1 * B2 + B1 * DB2

because of the monotonicity of B1 and B2. Sub-distributivity is then obvious.

Q. E. D.

REMARK: All the functions K(S) appearing in cases studied in earlier sections of this note and in [Sh] satisfy all the requirements of Theorem 2, and so have the serial selectability property. This implies the legitimacy of restrictions imposed in these cases on addition of elements to S, and makes a substantial part of the argumentation that would otherwise be required to justify those transformations unnecessary. Here we can make the general comment that discovery of metarules, such as the selection rule implied by serial selectability, always represents a significant step towards the automatization of transformational programming systems. Such metarules are also useful in manual construction of algorithms, both for proving algorithm correctness and selection of the transformations.

As an example illustrating the transformations which we have now justified, consider the following algorithm which tries to find a cycle S in a given graph G (cf. also [Sh] for a more detailed study of this algorithm):

```
S := {};
(while exists X in S st (forall Y in S : X(2) /= Y(1))
        or S = {})
    Z := arb* (G - S);
    S with:= Z;
end while;
```

First we must bring the predicate appearing in the while statement to the canonical form K(S) /= {}. (In what follows we will ignore the subpredicate S = {} which affects the selection of X only during the first iteration of the loop.) To do this, we rewrite the predicate as

```
{X in S : forall Y in S : X(2) /= Y(1)} /= {}
```

which can be transformed into

```
{X in S : X(2) notin {Y(1) : Y in S} } /= {}
```

and then into

```
S * {X : P2(X) notin P1[S]} /= {}
```

where P1 and P2 are the projections defined above. This allows us to rewrite the predicate as

```
K(S) /= {}
```

where

```
K(S) = S * (P2-1[P1[S]])c
```

K(S) has the form (2) with A(S) = S and B(S) = (P2-1[P1[S]])c. It follows easily from Lemma 3 that B(S) is in MSD-. Hence K(S) has the serial selectability property, which allows us to write the following version of the preceding algorithm:

```
S := {};
K' := {};
(while K' /= {} or S = {})
    Z := arb* {W in S - S st DK(S, {W}) * K(S) /= {} };
    K' := K' sym DK(S, {Z});
    S with:= Z;
end while;
```

However, in this case we have

```
DK(S, {W}) = DS * (P2-1[P1[S]])c
        sym S * P2-1[P1[S with W] - P1[S]]
        sym DS * P2-1[P1[S with W] - P1[S]]
```

and

```
DK(S, {W}) * K = S * P2-1[P1[S with W] - P1[S]]
        = {X in S : X(2) in P1[S with W] - P1[S]} .
```

But

```
P1[S with W] - P1[S] =
        if P1(W) in P1[S] then {} else {P1(W)} end
```

Hence

```
DK(S, {W}) * K =
        if W(1) in {Y(1) : Y in S} then {}
            else {X in S : X(2) = W(1)} end
```

Thus, since we can restrict DK * K to be nonempty, we obtain the following version:

```
S := {};
K' := {};
(while K' /= {} or S = {})
    Z := arb* {W in G - S :
        W(1) notin {Y(1) : Y in S}  and
        exists X in S st X(2) = W(1) };
    DK := if Z(1) in {Y(1) : Y in S} then {}
            else {X in S : X(2) = Z(1)} end;
    K' := K' sym DK;
    S with:= Z;
end while;
```

This version selects an edge Z whose source node is a target node of some edge in S but not a source node of any such edge. Note that the correctness of this selection heuristic is a formal consequence of our general selection rule, which also tells us that the new version can reach a successful termination if and only if the preceding version could.

REMARK: In certain cases K(S) may have the form (1) but involve also terms Ki(S) which are only monotone increasing in S. These terms must then be kept empty at all times or else the while loop in scheme (*) will never terminate. We can then use these terms to further restrict the selection of X in (*). Specifically, suppose that

```
K(S) = K0(S) + K1(S)
```

where K0(S) is monotone increasing in S and where K1(S) satisfies the requirements of Theorem 2. Using the results developed in section 2 concerning formal differentiation of predicates, we can then convert scheme (*) to the following variant of scheme (**):

```
S := {};
K1' := K1({});
assert K0({}) = {};
(while K1' /= {})
    X := arb* {W in Sc : K1(S) * DK1(S, {W}) /= {}
                and DK0(S, {W}) = {} };
    K1' := K1' sym DK1(S, {X});
    S with:= X;
end while
```

We now return to a more general discussion of formal principles. Assume that K(S) has the serial selectability property and let us re-consider scheme (**). Letting

(3)   L(S) = {W in Sc : K(S) * DK(S, {W}) /= {} }

we can transform scheme (**) into the following version, in which L(S) is also being maintained incrementally:

```
S := {};
K' := K({});
L' := L({});
(while K' /= {})
     X := arb* L';
     K' := K' sym DK(S, {X});
     L' := L' sym DL(S, {X});
     S with:= X;
end while;
```

If we assume that the derivative of L(S) can be expressed in a way which does not depend explicitly on K(S) then the only use of K' in this version of scheme (**) is to control the while loop. In the remaining part of this section we will consider the possibility of eliminating K' from the program altogether by using L(S) both in the selection of X and in controlling termination of the while loop.

An obvious condition that would permit this transformation is:

(4)   K(S) = {} iff L(S) = {}

which can be reformulated as follows (note that by definition K(S) = {} always implies L(S) = {}):

UNCONDITIONAL INCREMENTAL SELECTABILITY: K(S) is said to have the unconditional incremental selectability property if for each S subset E for which K(S) /= {} there exists X in Sc such that

K(S) not subset K(S + {X})

or, equivalently

K(S) * DK(S, {X}) /= {}

THEOREM 4: If K(S) has the unconditional incremental selectability property, then scheme (**) is equivalent to the following scheme:

```
(***) S := {};
      L' := L({});
      (while L' /= {})
           X := arb* L';
           L' := L' sym DL;
           S with:= X;
      end while;
```

Moreover, if scheme (***) is executed with arbitrary but deterministic selection of X then it always terminates and produces a solution of scheme (**).

PROOF: Scheme (***) is obtained from scheme (**) by replacing the test 'K(S) /= {}' by the test 'L(S) /= {}' (which is equivalent since we have assumed that K(S) has the unconditional incremental selectability property) and by eliminating dead code. Hence these schemes are equivalent. Our second assertion then follows from the fact that every execution of (***) must terminate, because an element X selected from L' is immediately removed from L' and is never added back to this set, because L' is disjoint from S to which this X is added.

Q. E. D.

Next we give a simple sufficient condition for K(S) to have the unconditional incremental selectability property:

THEOREM 5: Suppose that K(S) is a set-valued expression in S which satisfies the requirements of Theorem 2 and which also has the property that for each i <= r Bi(E) = {}. Then K(S) has the unconditional incremental selectability property.

PROOF: Let S be a subset of E such that K(S) /= {}. This implies that there exists i <= r such that Ki(S) /= {}. As in the proof of Theorem 2, for each X in Sc we obtain

    Ki(S) * DKi(S, {X}) = Ai(S) * DBi(S, {X})

But Bi(E) = {} by our assumption, and since Bi is monotone decreasing in S and DBi is sub-distributive in DS we obtain

    {} = Bi(E) = Bi(S + Sc) = Bi(S) - DBi(S, Sc)

Thus

    Bi(S) = DBi(S, Sc) subset
            +/ [DBi(S, {X}) : X in Sc]

and hence

    {} /= Ai(S) * Bi(S) subset
              +/ [Ai(S) * DBi(S, {X}) : X in Sc]
       = +/ [Ki(S) * DKi(S, {X}) : X in Sc]

Therefore there must exist X in Sc such that

    Ki(S) * DKi(S, {X}) /= {}

which implies that

    K(S) * DK(S, {X}) /= {}

and thus proves our theorem.

Q. E. D.

REMARK: Expressions K(S) satisfying the requirements of Theorem 5 have the property K(E) = {}, i.e. the universal set E is a solution to our problem. It is noteworthy that a converse statement can also be formulated, as follows: Suppose that K(S) satisfies the requirements of Theorem 3 and that K(E) = {}. This implies, for each i <= r,

    Ai(E) * Bi(E) = {}

and since Ai is monotone increasing, it follows that for each S subset E

    Ai(S) * Bi(E) = {}

we can therefore rewrite each subexpression Ki(S) as

    Ki(S) = Ai(S) * (Bi(S) - Bi(E))

and so obtain a representation for K(S) which satisfies the requirements of Theorem 5.

The significance of Theorem 5 is rather limited in the general case, because when it applies we already know one solution to our problem (namely E), and it is pointless to apply scheme (***) to obtain another solution. However, if we wish to compute a minimal (or smallest) solution of the equation K(S) = {} then even though E is known to be a solution we would like to apply scheme (***) to obtain those minimal solutions. Theorem 4 then raises the following important problem: Do all executions of scheme (***) yield minimal solutions of the equation K(S) = {}? This question is significant when an original specification asks for minimal solutions, and if its answer is positive we can use a deterministic version of (***) to solve the problem.

While in general this property need not hold (see below for an example), we will give a simple sufficient condition that guarantees it. It is hoped that similar conditions can be derived so as to extend the applicability of the deterministic scheme (***) to computation of minimal solutions of K(S) = {}, as this seems to be the best improvement in algorithm efficiency that our formalism was able to attain so far.

THEOREM 6: (a) Assume that L(S) has the property that the function

    N(S) = S + L(S)

is monotone increasing in S. Then, if K(S) has the unconditional incremental selectability property, the equation K(S) = {} has a smallest solution which will be produced by any deterministic execution of (***).

(b) Moreover, N(S) will be monotone increasing if the subexpression

    {W : K(S) * DK(S, {W}) /= {} }

is monotone increasing in S. Furthermore, this last condition must hold if K(S) has the form (1) and if in addition DBi is independent of S for each i <= r.

PROOF: The existence of a smallest solution of K(S) = {} follows by a standard fixpoint argument. Indeed, if one defines a sequence of sets by

```
S1 = {}
Sj+1 = Sj + L(Sj),    j := 1, 2 ...
```

then this sequence must converge to a set S0 which is the smallest solution of K(S) = {}. Next consider any deterministic execution of scheme (***), and suppose that the elements X1, X2 ... Xn have been selected from L' in order, so that S = {X1...Xn} is a solution of the equation L(S) = {} (and hence also of the equation K(S) = {}). By our assumption,

```
X1 in {} + L({})  subset  S0 + L(S0)  =  S0
```

so that

```
{X1}  subset  S0.
```

But this implies that

```
{X1} + L({X1})  subset  S0
```

which implies that X2, being an element of L({X1}) must belong to S0. Hence we have

```
{X1, X2}  subset  S0
```

and continuing in this manner we can show that S subset S0, and so by the minimality of S0 we conclude that S = S0. Since one always has

```
N(S) = S + {W : K(S) * DK(S, {W}) /= {} }
```

we conclude immediately that the monotonicity of the second set expression is sufficient to guarantee the monotonicity of N(S) itself. Finally, suppose that K(S) has the form assumed in the last assertion of the theorem. Then

```
{W : K(S) * DK(S, {W}) /= {} } =
     +/ [ {W : Ai(S) * DBi({W}) /= {} } : i in [1..r] ]
     = +/ [ DBi-1[Ai(S)] : i in [1..r] ]
```

which is obviously monotone increasing in S.

Q. E. D.

Theorems 4, 5 and 6 thus allow us to use deterministic workset algorithms to solve a fairly large class of subset construction problems.

As an example, consider the transitive closure scheme mentioned earlier. For this scheme we have

K(S) = F * P1-1[S] * P2-1[Sc]

and it follows easily from Lemma 3 that the monotone decreasing factor B(S) of K(S) is in MSO-. Hence K(S) has the incremental selectability property, which allows us to use scheme (**) to solve this problem. But since B(S) = {}, it follows from Theorem 5 that K(S) also has the unconditional incremental selectability property, which allows us to use scheme (***) instead of scheme (**). Moreover,

DK(S, DS) = P2-1[DS]

is independent of S, so that by Theorem 6 we know that there exists a smallest solution to our problem, and that any deterministic execution of (***) will yield this solution. To obtain such a deterministic algorithm, we write L(S) as follows:

L(S) = Sc * {W : K(S) * DK(S, {W}) /= {}}

= Sc * {W : F * P1-1[S] * P2-1[{W}] /= {} }

= Sc * P2[F * P1-1[S]]

= Sc * F[S]

Next we compute DL:

DL(S, {X}) =

{X} * F[S] (subset of L)

+ (F[{X}] - F[S]) * (Sc - {X}) (disjoint from L)

= {X} (subset of L)

+ F[{X}] * (Sc - L) (disjoint from L)

We thus obtain the following well known workset-oriented transitive closure algorithm:

```
S := S0;
L' := {W in Sc : U in S st [U, W] in F};
(while L' /= {})
    X := arb L';
    L' := L' less X
        + {Y in F{X} : Y notin S and Y notin L'};
    S with:= X;
end while;
```

NOTE: As a matter of fact, this transitive closure algorithm does not have exactly the form (***), because it initializes S to a nonempty set. It is interesting to note that this initialization can also be derived from our formal principles. Indeed, a direct translation from a specification of the transitive closure problem yields the following

scheme, which has the form (*) (and where the minimality condition is ignored):

```
S := {};
(while (S0 * Sc + F * P1-1[S] * P2-1[Sc]) /= {})
    X := arb* Sc;
    S with := X;
end while;
```

For this scheme we have

```
K(S) = S0 * Sc + F * P1-1[S] * P2-1[Sc]
```

which has the form (1) but involves two terms, both involving monotone decreasing factors belonging to MSD-. Hence K(S) has the serial selectability property. Moreover, it follows from Theorem 5 that K(S) has also the unconditional incremental selectability property, and K(S) is also easily seen to satisfy the requirements of Theorem 6. Thus, to obtain a deterministic version of scheme (***) for this problem, we compute L(S) to obtain:

```
L(S) = Sc * S0 + Sc * F[S]
```

Then we compute DL, obtaining

```
DL(S, {X}) = D(Sc * (S0 + F[S]))
     = {X}                    (subset of L(S))
       + F[{X}] - S - L       (disjoint from L(S))
```

This yields the following transitive closure algorithm:

```
S := {};
L' := S0;
(while L' /= {})
    X := arb L';
    L' := L' less X
        + {Y in F{X} : Y notin S and Y notin L'};
    S with:= X;
end while;
```

This last algorithm is in fact more elegant than the preceding one, since it simplifies the initialization of L'.

As another interesting application of our formal principles, consider the equivalence relation problem given in section 2. There we have

```
K(S) = A * Sc + B * Sc + S * I-1[Sc]
        + F-1[S] * G-1[S] * H-1[Sc]
```

where the functions I, F, G, H are as defined in section 2. K(S) obviously satisfies the requirements of Theorem 2 and so has the serial selectability property. Moreover it also satisfies the requirements of Theorems 5 and 6, so that we can obtain the smallest solution of 'K(S) =

{}' by the deterministic version of scheme (***). To achieve this, we first compute

```
K(S) * DK(S, {U}) =
     (A + B) * {U} +
     S * I-1[{U}] +
     F-1[S] * G-1[S] * H-1[{U}]
```

Hence,

```
L(S) = {U in Sc : U in (A + B) or U in I[S] or
                  U in H[F-1[S] * G-1[S]] }

       = Sc * (A + B + I[S] + H[F-1[S] * G-1[S]])
```

Next, we compute DL(S, {X}), assuming that X is in L(S):

```
DL(S, {X}) = {X}               (subset of L)
          + (Sc - {X}) * D(A + B + I[S] + H[F-1[S] * G-1[S]])
                               (disjoint from L)
```

Although the rule for differentiating a (nondisjoint) union (rule (5) of section 2) is somewhat complicated we can simplify it considerably in the example before us, noting that all the subexpressions A, B, I[S] and H[F-1[S] * G-1[S]] are monotone increasing in S. It is then easy to show that

```
D(A + B + I[S] + H[F-1[S] * G-1[S]]) =

     DA + DB + D(I[S]) + D(H[F-1[S] * G-1[S]])
        - (A + B + I[S] + H[F-1[S] * G-1[S]])
```

But

```
DA = DB = {}

D(I[S]) = I[{X}] - I[S]

D(H[F-1[S] * G-1[S]]) =
     H[D(F-1[S] * G-1[S])] - H[F-1[S] * G-1[S]]

   = H[F-1[{X}] * G-1[S]] + H[F-1[S] * G-1[{X}]]
     + H[F-1[{X}] * G-1[{X}]] - H[F-1[S] * G-1[S]]
```

All this implies

```
DL(S, {X}) = {X}
           + (I[{X}] + H[F-1[{X}] * G-1[S]] +
              H[F-1[S] * G-1[{X}]] + H[F-1[{X}] * G-1[{X}]]
              - L(S) - S)
```

Then, if we let X = [V, W] and use the definitions of the functions F, G, H and I, we obtain, after a few simplifications

```
DL(S, {X}) = {X}          (subset of L)
```

```
+ ({[W, V]} + {[V, Z] . [W, Z] in S}
  + {[Z, W] : [Z, V] in S} - L(S) - S)
          (disjoint from L)
```

Hence we can solve the equivalence relation problem by the following deterministic algorithm:

```
S := {};
L' := A + S;
(while L' /= {})
    [V, W] := arb L';
    L' := L' less [V, W]
         + ({[W, V]} + {[V, Z] : [W, Z] in S}
           + {[Z, W] : [Z, V] in S} - L' - S);
    S with:= [V, W];
end while;
```

This version can be further optimized e.g. by maintaining the inverse SINV of S also so that the setformers involved in the updating of L' can be cheaply constructed. Nevertheless, a comparison of the last version with the more cumbersome (and nondeterministic) algorithm derived in section 2, and the fact that the last version has been obtained from the problem specification in a purely formal manner, clearly show the great potential of our formalism.

Finally we give an example to which Theorems 5 and 6 do not apply. For this consider the following simple problem: Given a relation R on E x F, find a minimal subset S of E such that R[S] = F. It is easily seen that in this example we have

```
K(S) = R[S]c
```

which obviously satisfies the requirements of Theorem 2. However, for Theorem 5 to apply, we must have R[E] = F (which, in this example, is also a necessary condition for K to have the unconditional incremental selectability property). Thus, unless this condition is met, scheme (***) is not equivalent to scheme (**). Moreover, even if R[E] = F, in which case a deterministic version of scheme (***) can be used to compute a solution S to the equation K(S) = {}, there is no guarantee that this solution will be minimal. To see this, let E = F = {1,2}, and let

```
R = {[1, 1], [2, 1], [2, 2]} .
```

It is easy to check that for this example we have

```
L(S) = Sc * R-1[R[S]c]
```

so that 1 in L({}) and 2 in L({1}), which implies that the nonminimal solution S = {1, 2} can be obtained by some deterministic execution of scheme (***), which will first put 1 and then 2 into S.

In cases where there exists a smallest solution to K(S) = {} Theorem 6 gives a sufficient condition for that solution to be produced

by any deterministic execution of scheme (***). In other cases where such a smallest solution does not exist, but where the specification asks nevertheless for a minimal solution, we do not have as yet a similar condition that would guarantee that any solution produced by deterministic execution of (***) is minimal. This property has been observed, however, in most cases studied so far, and we conjecture that a relatively simple condition implying that minimality property can be stated.

Even if K(S) does not satisfy (4), but as long as it still has the serial selectability property, it remains possible to use scheme (***) to construct a subset S satisfying K(S) = {}, but in the narrower sense indicated by the following proposition:

PROPOSITION 7: Suppose that K(S) has the serial selectability property. Then any minimal subset S satisfying K(S) = {} can be computed by (the nondeterministic) scheme (***), but there may exist executions of (***) which produce subsets S that do not satisfy the equation K(S) = {}.

PROOF: Let S be a minimal subset of E satisfying K(S) = {}. Serial selectability for K then implies that S can be constructed by scheme (**) using a sequence of selections during which (4) holds. Hence, if scheme (***) is executed using the same sequence of selections it will also yield the set S. However there may exist other executions of (***) for which (4) does not hold, and since every execution of (***) is successful (see the proof of Theorem 4), there may exist executions of (***) producing sets S which do not satisfy K(S) = {}.

Q. E. D.

B I B L I O G R A P H Y

[DM] Dewar, R.B.K. and McCann, A.P., "MACRO SPITBOL - a SNOBOL4 Compiler", Software Practice and Experience 7(1977) 95-113

[De] Dewar, R.B.K., "The SETL Programming Language", To appear

[DSW] Dewar, R.B.K., Sharir, M. and Weixelbaum, E., "On Transformational Construction of Garbage Collection Algorithms", to appear.

[Ea] Earley, J., "High-Level Iterators and a Method for Automatically Designing Data Structure Representation Tech. Rep., Dept. of E.E. and Computer Science, University of California at Berkeley, 1974.

[Fo] Fong, A.C., "Inductively Computable Constructs in Very High Level Languages", Proc. 6th POPL, 1979.

[FU] Fong, A.C. and Ullman, J.D., "Inductive Variables in Very High Level Languages", Proc. 3th POPL, 1976.

[Pa] Paige, R., "Expression Continuity and the Formal Differentiation of Algorithms", Courant Comp. Sci. Report #15, Courant Institute 1979.

[PS] Paige, R. and Schwartz, J.T., "Expression Continuity and the Formal Differentiation of Algorithms", Proc. 4th POPL, 1977.

[Sh] Sharir, M. "Algorithm Derivation by Transformation", to appear as a Courant Institute Tech. Rept.

This book may be kept

FOURTEEN DAYS

A fine will be charged for each day the book is kept overtime.